# Introduction

In this document, I will explain the technical details, design choices, and approaches I used to implement the backend simulation and data-handling solution for Sedibelo Technologies. This solution focuses on security, performance, and scalability while addressing real-world challenges such as data deduplication, encryption, efficient file handling, secure communication, and idempotent API requests.

This solution demonstrates my ability to design and implement secure, efficient, and scalable backend systems. I followed best practices for encryption, API communication, file handling, and WebSocket interaction. The use of MongoDB ensured that the query for the Engineering department was optimized to handle large datasets. I also implemented rate-limiting, CSRF protection, and retry logic to ensure security and reliability.

The project showcases my expertise in handling complex data operations, secure communication, and real-time interaction while ensuring performance and scalability in modern web applications.

The solution utilizes Node.js for backend operations, file streaming for efficient data management, and WebSocket for real-time communication. Throughout the implementation, I adhered to modern development practices, ensuring robust security and smooth handling of large datasets.

## Task 1.1 - Simulating Secure Login Form Submission

This step simulates the submission of a login form securely to an HTTPS endpoint. It required implementing rate-limiting to prevent brute-force attacks and CSRF protection to ensure secure form submission. On successful login, the server returns a session token and a list of users, which needed to be encrypted and stored locally.

### Technical Approach

**HTTPS Communication** : HTTPS ensures secure transmission by encrypting data between the client and server. For this task, I used *axios*, a promise-based HTTP client, to make secure POST requests to the endpoint.

**CSRF Protection** : To mitigate Cross-Site Request Forgery (CSRF) attacks, I generated a CSRF token using Node.js's *crypto* module. The token would ideally be stored in a user's session and validated server-side on form submission. This token prevents unauthorized form submissions initiated from external sites.

**Rate-Limiting**: I implemented rate-limiting to limit the number of login attempts within a specific time window. This helps prevent brute-force attacks by blocking excessive login attempts from the same IP address. I used *express-rate-limit* middleware, which provides configurable *rate-limiting* functionality.

**Data Encryption and Storage**:I encrypted the users' data using the AES-256-CBC encryption algorithm. AES-256 provides strong encryption, ensuring that sensitive information is stored securely. The encrypted user list was saved to *users.json* for later use.

## Task 1.2 - Data Handling, Deduplication, and Performance Optimization

The task required reading the user data from *users.json*, removing duplicate entries, and assigning a UUID to each unique user. I also needed to generate a CSV file showing the number of times each user was duplicated. The solution had to be optimized to handle large datasets efficiently, simulating scenarios with millions of entries.

### Technical Approach

**Reading Large Files with Streaming** : Using file streaming minimizes memory usage when dealing with large datasets. I used *JSONStream* to read the contents of *users.json* incrementally. Unlike loading the entire file into memory, streaming allows processing large files chunk by chunk, preventing memory exhaustion.

**Deduplication Logic**: I utilized a Map data structure to store users, ensuring uniqueness based on their email address. This approach allows for O(1) time complexity for insertion and lookup operations, ensuring efficient deduplication even with millions of records.

**UUID Generation** : Each unique user was assigned a Universally Unique Identifier (UUID) using the *uuid* library. UUIDs ensure that every user has a unique, non-colliding identifier, even when distributed across different systems.

**CSV Generation for Duplicate Counts** : I used *fast-csv* to generate a CSV file containing the Name, Surname, and Number of Times Duplicated for each user. Fast-csv efficiently handles large datasets and generates the CSV incrementally, ensuring that the memory footprint remains low.

**Incremental File Writes** : To improve performance, the data was written in small chunks to avoid memory bottlenecks.

**Asynchronous Operations** : An additional performance advancement was to make sure that all file I/O operations were performed asynchronously to prevent blocking the event loop.

**Lazy Evaluation** : Lazy evaluation allowed the system to process the data only when needed, improving performance for large datasets.

## Task 1. 3 - Simulating Secure User Data Posting

The task involved securely posting user data from *uniqueUsers.json* to the specified endpoint with idempotency and retry logic for failed requests. Each user's data included the session token received during login.

### Technical Approach

**API Communication** : I used *axios* to post each user's data. The payload included all required fields, including the user's name, surname, designation, department, and unique UUID, along with the session token for authentication.

**Idempotency Handling** : Idempotency ensures that repeated API calls produce the same result without unintended side effects. Each user's UUID served as a unique identifier, preventing duplicate entries on the server.

**Retry Logic for Failed Requests** : I implemented retry logic using the *axios-retry* middleware. This ensured that in case of network failures or server issues, the request would be retried up to a certain limit, increasing the reliability of the system.

## Task 1.4 Engineering Department Reporting

From the list of unique users, I needed to identify those in the Engineering department and determine how many people (Mechanics and Mechanic Assistants) report to Michael Phalane. The query had to be optimized to handle large datasets using a NoSQL database such as MongoDB.

### Technical Approach

**Database Selection** : I used MongoDB due to its flexible schema design, which is ideal for handling hierarchical data structures. MongoDB's query capabilities also allow for efficient filtering and aggregation, even with large datasets

**Query Optimization** : I created a compound index on the *department* and *designation* fields to speed up the query.

**Aggregation Pipelines** : I used an aggregation pipeline to filter users in the Engineering department and group them by their manager (Michael Phalane).

**Sharding Consideration** : For scalability, the solution could be extended to use sharding in MongoDB, distributing the dataset across multiple nodes for improved performance.

## Task 1. 5 WebSocket Client

The goal was to create a WebSocket client that connects to the server, sends a string, and verifies that the returned string matches its reversed version.

### Technical Approach

**WebSocket Connection** : I used the *ws* library to establish a WebSocket connection to the provided endpoint. WebSockets provide full-duplex communication, enabling real-time data exchange.

**Verification Mechanism** : After sending a string to the server, the server returns the reversed string. I implemented a simple string comparison logic to verify that the returned string matches the reversed version of the original input.

## Task 1.6 Fibonacci Sequence Generator

The task required generating the Fibonacci sequence up to the nth number and returning the nth Fibonacci number.

### Technical Approach

**Iterative Approach for Efficiency** : I opted for an iterative approach to generate the Fibonacci sequence. This approach is more efficient than recursion, with a time complexity of O(n) and constant space complexity O(1).

**Handling Large Numbers** : For very large Fibonacci numbers, I ensured that the solution could handle BigInt values to prevent overflow.